Примена машинског учења у статичкој верификацији софтвера

Семинарски рад у оквиру курса
Методологија стручног и научног рада
Математички факултет

Лазар Ранковић, Немања Мићовић, Урош Стегић
lazar.rankovic@outlook.com, nmicovic@outlook.com, mi10287@alas.matf.bg.ac.rs

Абстракт

Верификација софтвера постаје све битнија дисциплина (реф). Класични приступи су добри, имају супер резултате (реф). Машинско учење постаје све популарније. Показаћемо преглед радова који примењују алгоритме машинског учења у циљу убрзања процеса верификације (реф).

Садржај

1 Uvod

Увод ћемо пред крај писати.

2 Верификација софтвера

Верификација софтвера је дисциплина развоја софтвера чији је циљ да се бави проверавањем да ли програм задовољава све унапред задате захтеве. Унапред задати захтев су спецификација свих жељених особина програма које се постављају пре процеса верификације. Највећа примена верификације софтвера је у оптимизацији кода и провере исправности.
Што се тиче исправности морамо направити разлику између тоталне исправности и делимичне исправности. Испитивање тоталне исправност захтева да се за све могуће улазе покаже заустављање програма, то на жалост није могуће, *"Halting problem"* је теорема која је доказана да је неодлучива. Поред тога постоји још једна теорема *"Rices theorem"* која гласи "Ни једно не тривијално семантичко својство програма није одлучиво". Према томе, због ових теорема у рачунарству је довољно показати да ће резултат евалуације програма бити валидна вредност, тачније неће се десити да програм врати вредност која није валидна.
Два приступа при верификацији софтвер су: *динамичка верификација* и *статичка верификација* којом ћемо се посебно бавити у наставку.

Динамичка верификација Динамичка верификација софтвера се врши у току извршавања програма и то најчешће скупом унапред припремљених тестова који морају бити задовољени. Очигледно је да због неисцрпне варијације могућих улаза овај вид тестирања нема за циљ валидацију програма, већ је циљ динамичке верификације проналажење грешка на неком не тривијалном скупу тестова.

Статичка верификација Статичка верификација софтвера подразумева анализу софтвера без његовог извршавања, тачније анализу кода применом неке од техника које ће бити описане у наставку. Анализа кода може бити ручна или аутоматизована. Ручна метода подразумева да човек проверава код, а аутоматизована подразумева описивање па чак и превођење кода на неки од математичких језика, изабране математичке теорије. Најчешћа употреба статичке верификације је оптимизација кода при превођењу. ======= *Верификација софтвера* је дисциплина развоја софтвера која за циљ има провераву да ли програм задовољава све унапред задате захтеве. Ти захтеви су представљени спецификацијом свих жељених особина програма и дефинишу се пре процеса верификације. Највећа примена верификације софтвера је у оптимизацији кода и провери исправности.

Потребно је направити разлику између тоталне и делимичне исправности. Испитивање тоталне исправности захтева да се за све могуће улазе покаже заустављање програма. Доказ заустављања у општем случају није могуће извести [14]. Такође, испитивање нетривијалних семантичких својстава је неодлучив проблем [1]. Према томе, у рачунарству је довољно испитати делимичну исправност софтвера, тј.

довољно је показати да ће резултат извршавања програма бити валидна вредност.

Два приступа при верификацији софтвера су *динамичка верификација* и *статичка верификација* [7]

- **Динамичка верификација**
  Овај вид верификације се врши у току извршавања програма и то најчешће скупом унапред припремљених тестова који морају бити испуњени. Очигледно је да због неисцрпне варијације могућих улаза, овај вид тестирања нема за циљ валидацију програма, већ је циљ динамичке верификације проналажење грешка на неком не тривијалном скупу тестова.
- **Статичка верификација**
  Статичка верификација софтвера подразумева анализу софтвера без његовог извршавања, тачније анализу кода применом неке од техника које ће бити описане у наставку. Анализу кода може обављати човек, а може се и аутоматизовати. Аутоматизација подразумева описивање (па чак и превођење) кода језиком изабране математичке теорије.

У наставку ће бити више речи о статичкој верификацији. Биће описани механизми анализе кода, технике верификације и чести проблеми овог типа верификације.

3 Технике статичке верификације

Апстрактна интерпретација је је теорија семантичке апроксимације чија је идеја да направити нову семантику над програмским језиком тако да се конкретан програм увек завршава. Тако се анализа програма врши над апстрактном семантиком да би добили апроксимацију над целом семантиком. Коришћење апстрактне интерпретације се омогућава помоћу две функције: функције која пресликава конкретне вредности у апстрактне вредности и функције која слика апстрактне вредности у конкретне вредности. Неизбежно је заобићи губљење података при пресликавању из конкретних вредности у апстрактне вредности јер је циљ показати да се над апострактном семантиком програм завршава. Користећи овај математички оквир је релативно лако показати да ако се програм завршава у новој семантици програм ће бити коректан и у стварној семантици.

Симболичка анализа је метод статичке анализе који анализира програмске вредности који могу да се мењају. Овај метод има за циљ да изведе математички модел који прецизно описује израчунавање, заправо може се посматрати као нека врста компајлера који преводи програм у симболичке изразе. Квалитет алгебарских система као што су (Axiom, Derive, Macsyma, Maple, Mathematica, MuPAD, and Reduce) је веома битан јер квалитет овог начина анализе у великој меримо зависи од паметних алгебарских упрошћавања.

Проверавање ограничених модела (енг. Bounded model checking)
Проверавање ограничених модела је техника верификације која се највише користи у индустрији полуповодника, тачније верификација логичкх кола. Укратко речено смисао је да се логичка кола опишу исказном логиком. Следећи корак је провера задовољивости добијене

исказне формуле. Испитивање задовољивости формула је НМ-тежак проблем, и за решавање овог питања користе се сат решавачи. Ефикасност сат решавача је од кључног значаја за ову технику. Ова техника је такође примењива и за анализу софтвера, један од начина примене је посматрање извршавања целокупног програма као скупа стања, односно као један коначни аутомат у ком се прелази из стања у стање. Ако се тако посматра програм могуће је описати сва стања искасном логиком затим повезати сва стања и тако добијену формулу пустити у сат решавач. Резултат сат решавача је може бити формула је задовољива сто би значило да је програм коректан или ако је формула незадовољива резултат ће бити контрапример којим се показује да програм није коректан и може представљати основу за дебаговање.

Литература: A Survey of Static Program Analysis Techniques [15]
A Survey of Automated Techniques for Formal Software Verification [5]
======= Претходним поглављем су дефинисани основни појмови верификације софтвера. Предочено је да је немогуће у општем случају испитати заустављање програма и анализирати нетривијална семантичка својства. Ово поглавље ће дати увид у аутоматизоване технике статичке анализе.

3.1 Апстрактна интерпретација

Апстрактна интерпретација је теорија семантичке апроксимације чија је идеја да изгради нову семантику над програмским језиком тако да се конкретан програм увек завршава. Тако се анализа програма врши над апстрактном семантиком да би се добила апроксимација над целом семантиком. Коришћење апстрактне интерпретације се омогућава помоћу две функције: функције која пресликава конкретне вредности у апстрактне вредности и функције која слика апстрактне вредности у конкретне вредности. Неизбежно је заобићи губљење података при пресликавању из конкретних вредности у апстрактне вредности јер је циљ показати да се над апострактном семантиком програм завршава. Користећи овај математички оквир је релативно лако показати да ако се програм завршава у новој семантици програм ће бити коректан и у стварној семантици.

3.2 Симболичка анализа

Симболичка анализа је метод статичке анализе који анализира програмске вредности који могу да се мењају. Овај метод има за циљ да изведе математички модел који прецизно описује израчунавање, заправо може се посматрати као нека врста компајлера који преводи програм у симболичке изразе. Квалитет алгебарских система као што су (Axiom, Derive, Macsyma, Maple, Mathematica, MuPAD, and Reduce) је веома битан јер квалитет овог начина анализе у великој мерим зависи од паметних алгебарских упрошћавања.

3.3 Проверавање ограничених модела

Проверавање ограничених модела (енг. Bounded model checking) је техника верификације која се највише користи у индустрији полупроводника, тачније верификација логичкх кола. Укратко речено смисао је да се логичка кола опишу исказном логиком. Следећи корак је

провера задовољивости добијене исказне формуле. Испитивање задовољивости формула је НМ-тежак проблем, и за решавање овог питања користе се сат решавачи. Ефикасност сат решавача је од кључног значаја за ову технику. Ова техника је такође примењљива и за анализу софтвера, један од начина примене је посматрање извршавања целокупног програма као скупа стања, односно као један коначни аутомат у ком се прелази из стања у стање. Ако се тако посматра програм могуће је описати сва стања искасном логиком затим повезати сва стања и тако добијену формулу пустити у сат решавач. Резултат сат решавача је може бити формула је задовољива сто би значило да је програм коректан или ако је формула незадовољива резултат ће бити контрапример којим се показује да програм није коректан и може представљати основу за дебаговање.[15][5].

Овим поглављем смо направили кратак преглед најважнијих техника аутоматизоване статичке анализе. Показује се да наведене технике дају неку врсту формалне гаранције квалитета софтвера. Апстрактна семантика је уопштени начин рада кад се прави програм за анализу софтвера, тако да се сви остали програми који спроводе анализу могу бити посматрани као инстанце апстрактне семантике. Симболичка анализа је техника којом се изводи прецизна карактеризација програмских својстава на параметарски начин, а техника проверавања ограничених модела је моћна за октривање једноставних грешака али на жалост нису у стању да докажу чак ни дубоке петље.
У следећем поглављу ћемо објаснити основне аспекте области машинског учења.

4 Машинско учење

У претходним поглављима је описана статичка верификацију софтвера. Показана је важност те области и изложене су технике верификације. Ово поглавље ће приближити област машинског учења и описаће главне аспекте ове дисциплине,

4.1 Основе машинског учења

Дефиниција 1. *"За програм кажемо да учи из искуства E кроз обављање задатка T са мером квалитета P, ако повећањем искуства E расте мера P за обављен задатак T."*

— Tom M. Mitchell *[10]*

Машинско учење се може посматрати као област рачунарства која се бави анализом алгоритама који генерализују. Са практичног аспекта, генерализација може значити уопштавање закона над датим подацима.

Три најзначајније подобласти машинског учења су: *надгледано учење*, *ненадгледано учење* и *учење условљавањем*. Подаци из којих алгоритми машинског учења уче, могу бити обележени, необележени и могу се генерисати у фази учења. Оваква природа података је основ за разликовање три наведене подобласти.[10].

Међу многим проблемима над којима су често примењивани алгоритми машинског учења, истактнути су проблем регресије и проблем класификације. Под проблемом класификације се подразумева испитивање датог објекта и одређивање класе којој он припада на

основу његових својстава (атрибута). Типичан пример класификације је одређивање порекла тумора на основу његове величине. Проблем регресије представља предвиђање понашања непрекидне променљиве. Пример регресије је предикцију цене стамбеног објекта на основу његове величине, броја соба и разних других релевантних карактеристика.

Пре примене машинског учења потребно је проучити проблем који се решава, уочити његове специфичности и припремити и анализирати податке из којих ће алгоритми учити. Након детаљне анализе, врши се одабир одговарајућег математичког модела. Изабрали модел се даље тренира над подацима. Тренинг се понавља довољан број пута при чему у свакој итерацији модел евалуира тј. мери се грешка коју тај модел прави. Након сваког мерења, у зависности од алгоритма, врши се корекција модела у циљу минимизације грешке.

У даљем тексту ће бити приказани конкретни алгоритми који су релевантни за процес верификације и дискутоваће се о њиховим својствима.

4.2 Значајност машинског учења

Конвенционалан начин решавања проблема у рачунарству се своди на формално дефинисање низа корака који улазне параметре трансформишу не би ли дошли до резултата. Овакав приступ је користан у ситуацијама када је потребно решити проблеме који су човеку изазовни, као што су компликоване рачунске операције, сортирање великих низова и томе слично. Поставља се питање: како написати програм који би обављао задатке које човек свакодневно лако обавља? На пример, да ли је могуће написати програм који би био у стању да препознаје објекте са фотографија?

Рачунарски вид (енг. computer vision) је дисциплина која се бави овим проблемом.[4]. Алгоритми који су примењивани пре раста популарности машинског учења нису показали значајне резултате. Могли су да генеришу једноставне геометријске моделе који нису давали задовољавајуће резултате. Дубоке неуронске мреже су алгоритами машинског учења који су показали значајне напретке у овој области [2].

Класификација дела програма на валидна стања и она која могу резултовати грешком је од кључног значаја за ефикасност алата за верификацију [3] [8]. Конкретним проблемима и њиховим решењима ћемо се бавити у наредним поглављима.

4.3 Технике машинског учења

О општој слици примене алгоритама машинског учења је било више речи у уводном делу овог поглавља. Како је проблем класификације централни проблем над којиме се примењује машинско учење у статичкој верификацији, биће представљена два алгоритма која решавају тај проблем. Зарад потпуности, биће описан и један алгоритам решавања регресионих проблема.

Линеарна регресија

Као што је речено у уводном делу, регресиони проблем представља предвиђање циљне променљиве непознате инстанце на основу осталих

њених атрибута. Нека је y_i циљна променљива, а $\vec{x} = (x_1, x_2, ..., x_n)$ вектор атрибута. У примеру предикције вредности куће то могу бити број соба, квадратура куће итд. Инстанца из скупа података ће онда бити $(\vec{x}_i, y_i)$. Модел линеарне регресије, параметризован вектором w, који описује законитост је следећи:

$$h(\vec{x}_i) = w^T \cdot \vec{x}_i \tag{1}$$

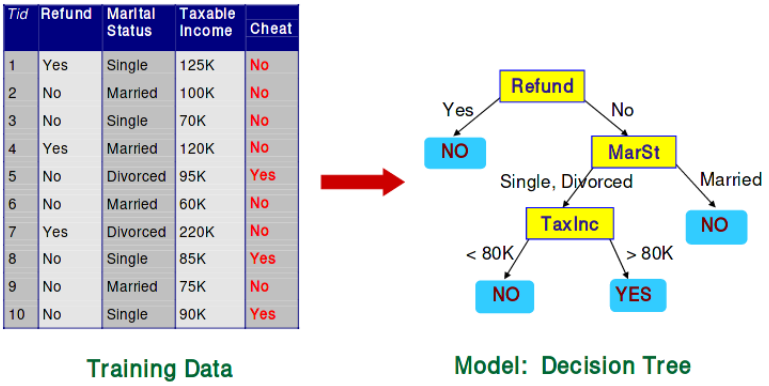
Грешка коју модел прави је потребно представити погодним избором функције грешке $L(w)$. Чест избор ове функције је средњеквадратна грешка коју модел прави над свим инстанцама из тренинг скупа.

$$L(w) = \frac{1}{N} \sum_{i=1}^{N} (h(\vec{x}_i) - y_i)^2 \tag{2}$$

Тренинг се врши тако што се одређеном оптимизационом техником минимизује функција грешке по параметрима w.

Стабла одлучивања

Стабла одлучивања представљају један од основних метода класификације. Употребу стабала одлучивања оправдава њихова висока интерпретабилност [12]. У листовима стабла одлучивања се налазе вредности циљне променљиве, односно у случају класификације, класе којима инстанца може припасти. Унутрашњи чворови стабла представљају атрибуте по којима се врши подела. Када су ти атрибути категоричког типа, потомци датог чвора су добијени из свих могућих вредности које тај категорички атрибут може имати. У случају да је атрибут некатегоричког типа, најчешће се врши подела могућих вредности на дисјунктне интервале тако да свако дете тог чвора одговара једном од интервала. Слика 1 приказује једно могуће стабло одлучивања добијено на основу података.

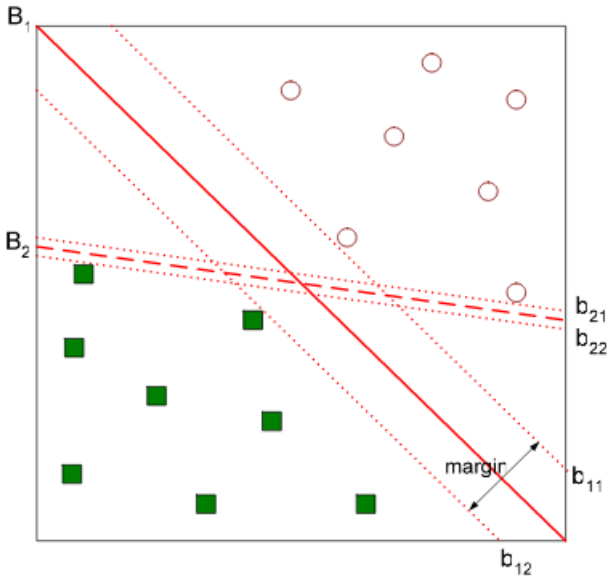


Слика 1: Стабло одлучивања

Метода потпорних вектора

Проблем класификације се може разматрати у следећем контексту. Инстанце које се класификују су представљене тачкама у неком високодимензионалном простору. Бинарни класификатор је хиперраван која дели простор на два дела, тако да се у једном делу простора нађу све инстанце које припадају једној класи, а у другом делу ће се наћи оне које припадају другој класи. Раздвајајућих хиперравни може бити више, па је зато потребно одабрати хиперраван која боље описује поделу међу подацима [11].

Маргина класификације је најмање растојање између тачака које се налазе у различитим потпросторима и бинарног класификатора. Слика 2 приказује хиперравни B_1 и B_2. Прва хиперраван боље раздваја податке. Маргина (b_11, b_12) је значајно већа од маргине (b_{21}, b_{22}) и то је оно што први класификатор чини знатно бољим.



Слика 2: Приказ различитих хиперравни

Максимизацијом маргине се добија класификатор који боље описује поделу. Зарад конвенције, проблем максимизације се своди на проблем минимизације, те се добија следећи оптимизациони проблем:

$$\min_{w,w_0} \frac{\|w\|^2}{2} \tag{3}$$

Линеарна регресија, стабла одлучивања и метод потпорних верктора су приказани у овом поглављу због својих примена у проблемима статичке верификације. Поглавље шест се бави овим применама. Следеће поглавље ће представити релевантне проблеме верификације и даће основ за примену ових метода.

5 Одабрани проблеми статичке верификације

До сада смо видели стандардне проблеме и технике статичке верификације и машинског учења. У овом поглављу ћемо издвојити значајне проблеме верификације на које су, применама алгоритама машинског учења постигнути значајнији резултати.

Статичка верификација мора бити у стању да разликује позитивна стања програма од негативних. Негативна су она која доводе програм до грешке. *Интерполантама* (енг. interpolants) називамо предикате који раздвајају позитивна од негативних стања. У статичкој верификацији се коришћењем оваквих интерполанти гради даљи доказ. Показано је да се ове интерполанте могу интерпретирати као бинарни класификатори. Проблем који се овде јавља је генерисање интерполанти, тј проналажење одговарајућег класификатора [13]. У делу 6 детаљније је описан приступ коришћен у [13].

Поред итерполанти, могуће је препознати нетривијална својства програма која даље резултују грешком. Грађењем *класификатора нетачне инваријанте* (енг. False Invariant Classifier) је могуће рангирати својства програма по томе колику вероватноћу за грешком та својства проузрокују. Одређивање нетривијалног својства датог програма је у општем случају неодлучив проблем [14, 3].

Код апстрактне интерпретације је остварив баланс између прецизности изгенерисане инваријанте и скалабилности система за верификацију. Овај баланс је последица детаљне анализе апстрактног синтаксног стабла. Одабир инваријанте је тежак проблем и показано је да се може утврдити тестирањем [13, 8].

Проблеми које смо представили овим поглављем су решена користећи одговарајуће технике машинског учења. У наредом поглављу ћемо се бавити тим решењима, даћемо увид у начине на који су та решења примењена и покушати да одговоримо на питање како наставити усавршавање тих техника.

6 Неке примене техника машинског учења у статичкој верификацији

Ово је есенција. Одабирају се проблеми из претходног поглавља и показује се како се решава. Прво иде неки уводни део, онда из литературе се покупе те технике и таксативно се наводе (принцип проблем-решење).

Проналажење интерполанти Неформално говорећи, интерполанта представља предикат који раздваја позитивна стања програма од негативних. Примена машинског учења у проналажењу интерполанти огледа се у добијању модела који представља саму интерполанту. У делу 6 изложене су основе из [13] базиране на методу потпорних вектора, док је у делу 6.1 изложен приступ из рада [8] базиран на стаблима одлучивања. Експериментални резултати показали су да приступи базирани на машинском учењу јесу упоредиви са традиционалним техникама.

Проналажење интерполанти користећи метод потпорних вектора Нека су A и B формуле у теорији линеарне аритметике [9].

$$\phi ::= w^T x + d \geq 0 \mid true \mid false \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \tag{4}$$

При чему је $\vec{w} = (w_1, ..., w_n)^T \in R^n$ вектор константи у простору R^n; $\vec{x} = (x_1, ..., x_n)^T$ вектор променљивих из простора R^n.

Дефиниција 2. *Интерполанта за пар формула (A, B) тако да* $A \wedge B \equiv \bot$ *је формула* I *која задовољава* $A \Rightarrow I, I \wedge B \equiv \bot$ *при чему формула* I *садржи само променљиве које се јављају у формулама* A *и* B.

На слици 3 приказан је програмски код који ће бити коришћен као илустрација. Функција непознат број пута инкрементира променљиве x и y, потом их декрементира све док променљива x не постане 0. Коначно, уколико је $y \neq 0$ онда програм одлази у стање грешке. Приметимо да је инваријанта $x = y$ довољна да се докаже да програм никад неће доћи у стање грешке.

```
foo( )
{
1:  x = y = 0;
2:  while (*)
3:   { x++; y++; }
4:  while ( x != 0 )
5:   { x--; y--; }
6:  if ( y != 0 )
7:    error() ;
}
```

Слика 3: Пример кода

Претпоставимо да је функција `foo()` извршила на следећи начин (у заградама су хронолошки наведени линије инструкција): $(1, 2, 3, 2, 4, 5, 4, 6, 7)$ који води у стање грешке. Поделимо ток на два скупа, A и B и пронађимо интерполанте за наведени ток.

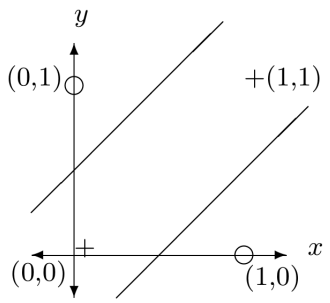
Скуп A садржи вредности x и y које се добијају након извршавања линија 1, 2 и 3. У скупу B се налазе оне вредности x и y које би се добиле уколико би програм извршио линије 4, 5, 6 и 7 чиме би програм дошао у завршно стање.

Имамо да $A \wedge B \equiv \bot$ при чему важи:

$$A \equiv x_1 = 0 \wedge y_1 = 0 \wedge if\_then\_else(b,\ x = x_1 \wedge y = y_1,\ x = x_1 + 1 \wedge y = y_1 + 1)$$
$$B \equiv if\_then\_else(x = 0,\ x_2 = x \wedge y_2 = y,\ x_2 = x - 1 \wedge y_2 = y - 1) \wedge x_2 = 0 \wedge \neg(y_2 = 0)$$

A представља скуп достижних стања док B представља скуп стања која воде у стање грешке. Интерполанта је доказ да су скупови A и B дисјунктни и изражава се користећи заједничке променљиве из скупова A и B. Затим, помоћу доказивача теорема се рачунају вредности за (x, y) које задовољавају формуле A и B [13].

Добијене вредности представљају скуп инстанци над којим се може тренирати класификациони модел (попут логистичке регресије или потпорних вектора). Позитивне инстанце представљају вредности



Слика 4: Класификација у тражењу интерполанти

Табела 1: Добијене интерполанте на неким од познатијих тест примера у области.

Датотека	**Време (с)**	**Интерполанта**
`f1a`	0.022	`((y = 1 | x <= 0) & x = 1) | (y = 0 & (y = 1 | x <= 0))`
`ex1`	0.021	`xa + 2*ya >= 0 | xa + 2*ya >= 5 | xa + 2*ya >= 5`
`f2`	0.20	`y <= 3*x | y <= 3*x + 1 | y <= 3*x + 1`
`nec1`	није доступно	Није пронађена
`nec2`	0.018	`x < y` (исто)
`nec3`	0.016	`y <= 9` (исто)
`nec4`	0.021	`(x = y | y = 0) | (y = x) | (y = x)`
`nec5`	0.018	`s >= 0` (исто)
`pldi08`	0.017	`y > x`
`fse06`	0.017	`y + x >= 0 & y >= 0 & y >= 0 & y >= 0`

променљивих које задовољавају формулу A и аналогно, негативне инстанце представљају вредности променљивих које задовољавају формулу B.

Слика 4 приказује вредности променљивих (x, y) за A као плусеве (тачке $(0, 0)$ и $(1, 1)$) и B као кружиће (тачке $(1, 0)$ и $(0, 1)$). Приказани модел је добијен коришћењем метода потпорних вектора. Резултујуће праве одговарају једначинама:

$$e_1 : 2y = 2x + 1$$
$$e_2 : 2y = 2x - 1$$

Интерполанта која се одавде може извести је

$$2y \leq 2x + 1 \wedge 2y \geq 2x - 1$$

Овај предикат представља инваријанту чијим доказивањем се показује да програм не може доћи у стање грешке. Једноставнија интерполанта $x = y$ се може добити транслирањем добијених правих што ближе позитивним истанцама, докле год се одржава сепарабилност позитивних и негативних инстанци.

Табела 1 приказује резултате из [13] на неким од познатих примера. Интерполанте које су означене са *исто* су интерполанте које су добијене користећи решавач OpenSMT.

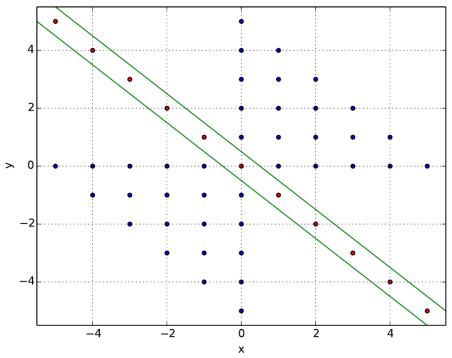
6.1 Проналажење интерполанти користећи стабла одлучивања

Интерполанте се могу извести и другим методима машинског учења. Рад [8] илуструје приступ који користи стабла одлучивања. За програмски код се генеришу позитивне и негативне инстанце над којима се гради стабло одлучивања користећи похлепни алгоритам. Правила добијена у стаблу се трансформишу у формулу која се потом проверава да ли је инваријанта користећи SMT решавач.

Резултати су показали да једноставни похлепни алгоритам који гради стабло даје и једноставне формуле за интерполанте. Стабло је лако научило комплексне бинарне инваријанте као једноставне коњункције.

Слика 5 приказује пример програма и његова стања која се могу добити на основу покретања самог програма. Добра стања можемо добити пратећи претпоставке (линија 2), бележећи ток променљивих и провером да ли је испуњен услов $x \neq 0$ са линије 12. Лоша стања можемо добити игноришући услов са линије 2. На пример, тачка $(-2, -2)$ тачка $(-4, -4)$ представљају лоша стања.

```
1  var x, y: Int;
2  assume x = 0 ∧ y ≠ 0;
3
4  while (y ≠ 0) {
5    if (y < 0) {
6      x := x − 1; y := y + 1;
7    } else {
8      x := x + 1; y := y − 1;
9    }
10 }
11
12 assert x ≠ 0;
```



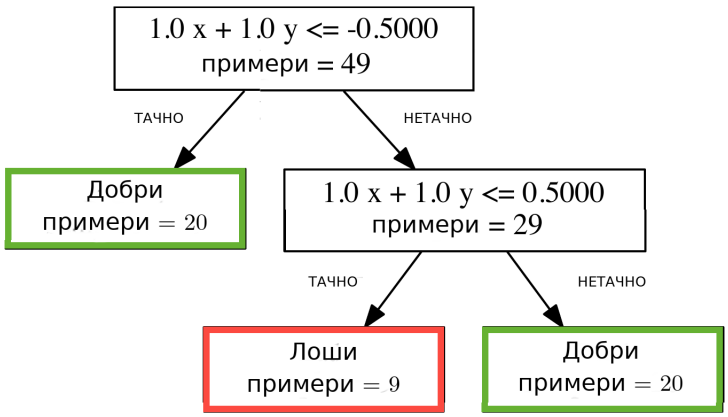
Слика 5: Пример програма. Лева страна приказује код, десна страна садржи добра стања (плаве тачке) и лоша стања (црвене тачке).

Слика 6 приказује стабло добијено применом алгоритма описаног у [8]. Добијени алгоритам је сложености $O(mn\log(n))$, где је m број атрибута а n број инстанци.

Грађење класификатора нетачне инваријанте Грађење класификатора нетачне инваријанте [3] представља приступ којим се користећи класификатор рангирају својства програма по њиховој вероватноћи да проузрокују грешку. Програмеру се приказује листа пронађених програмских својстава која треба да скрати простор претраге у потрази за грешком. У раду су коришћене две техника класификације машинског учења, метода потпорних вектора и метода стабла одлучивања. Као помоћ при проналажењу својстава програма, користи се Daikon динамички детектор инваријанти [6].

Daikon детектује својства специфичних делова програма попут улаза и излаза из процедура. За скаларне променљиве x, y и z, и константе a, b и c, неки примери својстава су:

- једнакост са константом ($x = a$);



Слика 6: Стабло одлучивања добијено за пример са слике 5.

- провера опсега ($a \leq x \leq b$);
- линеарне везе ($z = ax + by + c$);
- уређење ($x \leq y$);
- функције ($y = fn(x)$).

За секвенцијалне промењиве (низови, листе) нека својства су:

- минимум и максимум;
- лексикографско уређење;
- својства која важе за све елементе;
- припадност ($x \in y$).

Daikon може пронаћи и импликације попут *Ако је* $p \neq null$ *онда* $p.value > x$ и дисјункције попут $p.value > c \vee p.left \in T$.

Добијена својства је потребно превести у векторе како би се омогућила примена алгоритама машинског учења. Један пример кодирања добијених својстава је приказан на слици 7. У раду је кодирање изведено у векторе димензије 388 јер се показало да су коришћени алгоритми успевали да игноришу ирелевантне атрибуте те висока димензионалност није сметала.

Na slici 8 je prikazan programski kod, a na slici 9 lista programskih svojstava koji mogu biti potencijalni problemi. Dobijena lista programskih svojstava predstavlja klasifikovane instance koje potencijalno otkrivaju grešku.

7 Закључак

Радови приказани у делу 6 показали су да област машинског учења може пронаћи примену у области статичке верификације софтвера. Добијени резултати су били барем упоредиви са другим приступима, а у неким случајевима и доста бољи. Неки од проблема који се јављају при употребни алгоритама машинског учења јесу неинтерпретабилност добијеног модела и неегзактна предвиђања које модел

Property	Equation				Variable type			# vars	Score
	$\leq$	$=$	$\neq$	$\subseteq$	int	double	array		
out[1] $\leq$ in[1]	1	0	0	0	0	1	0	2	19
$\forall$ i: in[i] $\leq$ 100	1	0	0	0	0	1	0	1	16
in[0] = out[0]	0	1	0	0	0	1	0	2	15
size(out) = size(in)	0	1	0	0	1	0	0	2	13
in $\subseteq$ out	0	0	0	1	0	0	1	2	12
out $\subseteq$ in	0	0	0	1	0	0	1	2	12
in $\neq$ null	0	0	1	0	0	0	1	1	10
out $\neq$ null	0	0	1	0	0	0	1	1	10
Model weights	7	3	2	1	4	6	5	3	

Слика 7: Пример кодирања програмских својстава у вектор.

```
// Return a sorted copy of the argument.
double[] bubble_sort(double[] in) {
  double[] out = array_copy(in);
  for (int x = out.length - 1; x >= 1; x--)
    // lower bound should be 0, not 1
    for (int y = 1; y < x; y++)
      if (out[y] > out[y+1])
        swap(out[y], out[y+1]);
  return out;
}
```

Слика 8: Пример програмског кода.

Ranked properties	Fault-revealing?
out[1] $\leq$ in[1]	Yes
$\forall$ i: in[i] $\leq$ 100	No
in[0] = out[0]	Yes
size(out) = size(in)	No
in $\subseteq$ out	No
out $\subseteq$ in	No
in $\neq$ null	No
out $\neq$ null	No

Слика 9: Листа понуђених програмских својстава

врши. Проблем интерпретабилности је превазиђен стаблима одлучивања [8, 13] која су позната да дају интерпретабилне моделе, док је проблем неегзактног предвиђања ублажен у раду [3] где се као резултат даје листа програмских својстава које човек анализира. Уколико је неко својство погрешно класификовано, неће проузроковати велику грешку.

Литература

[1]

[2] Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876, 2015. Withdrawn.

[3] Yuriy Brun. Finding latent code errors via machine learning over program executions, 2004.

[4] Christopher M. Brown Dana H. Ballard. *Computer vision*. Prenice-Hall, Inc., 1982.

[5] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[7] Milena Vujosevic Janicic. Regresiona verifikacija softvera korišćenjem sistema lav.

[8] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

[9] Daniel Kroening and Ofer Strichman. *Linear Arithmetic*, pages 111–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] Tom M. Mitchell. *Machine Learning*, volume 1. McGraw Hill, 1997.

[11] John Shawe-Taylor Nello Christianini. *An introduction to support vector machines*, volume 1. Cambridge university press, 2000.

[12] David Landgrebe S. Rasoul Safavian. A survey of decision tree classifier methodology, 1991.

[13] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers.

[14] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[15] Wolfgang Wögerer and Technische Universität Wien. A survey of static program analysis techniques, 2005.